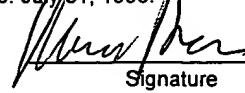


Express Mail Label No. 109093US

I hereby certify that I deposited this paper with the U.S. Postal Service as first class mail addressed for delivery to BOX PATENT APPLICATION, Assistant Commissioner for Patents, Washington, DC 20231, on this date: July 31, 1998.

Ronald Abramson
Name (Print)



7/31/98
Date

U.S. Patent Application:

**Object Oriented System for Managing
Complex Financial Instruments**

Inventors:

**Michael DeAddio
of White Plains, New York**

and

**Axel Kramer
of New York, New York**

Patent to be Assigned to:

**J.P. Morgan & Co. Incorporated
60 Wall Street
New York, New York 10260-0060**

Attorneys:

**Ronald Abramson, Reg. No. 34,762
Hughes Hubbard & Reed LLP
One Battery Park Plaza
New York, NY 10004-1482
(212) 837-6404**

OBJECT ORIENTED SYSTEM FOR MANAGING
COMPLEX FINANCIAL INSTRUMENTS

BACKGROUND OF THE INVENTION

FIELD OF INVENTION

This invention generally relates to the field of systems for data processing in the
5 financial services industry, and more particularly to systems for managing substantial
portfolios of derivatives and other complex financial instruments now widely used in
that industry.

DESCRIPTION OF THE RELATED ART

BACKGROUND OF THE INVENTION

There are several major domains of interest in the design and implementation of
10 modern risk management systems. These include:

- modeling of valuation methodologies
- modeling of financial products
- modeling of market environment information
- frameworks for risk analysis
- frameworks for persistence

Much thought and effort has been put into the study of valuation/pricing models
for the financial services industry. These studies are often of a highly theoretical and aca-
demic nature. While the continued development of pricing models is absolutely essential
for the evolution of the financial business it does not supply the entire solution.

20 As a result of our experience in this field, we have concluded that the financial
services industry cannot just concentrate on the valuation models alone. In practice, a
typical financial institution at any time will have positions in significant numbers of fi-
nancial instruments, the terms of which may vary significantly from one another, even
within a single series of instruments. Separate and apart from the science of pricing and
25 valuing individual instruments, is the practical problem of managing such a number of

dissimilar instruments in a consistent manner. Another issue is adjusting the processing of those instruments in a consistent and coordinated way as the pricing and valuation techniques themselves evolve and change.

The ability of a firm's data processing systems to deal with such a number and 5 variety of instruments is essential in order to support reliable and consistent financial and regulatory reporting, which is a key operating requirement for virtually all firms in this industry. Without such a system, the pricing results for similar instruments may vary inexplicably, portfolio valuations may be unreliable, and implementing new or modified pricing, valuation or processing techniques may require cumbersome instrument-by-10 instrument adjustments which can become impracticable.

In order to solve such problems, we must study generic computer science methodologies that bring pragmatic systems considerations to light, which may then be applied. In addition to providing a new system architecture, our solution involves a formalized approach to the implementation of pricing and risk management systems, which 15 we believe is also essential to the success of the financial business in this area.

BRIEF SUMMARY OF THE INVENTION

It is an object of the present invention to address the issues of consistency, manageability and modifiability described above by applying object oriented design strategies and patterns to the modeling and processing of financial products (also referred to as 20 financial instruments) with an emphasis on derivative products. It is a further object of the invention to provide means to specify financial instruments in a consistent manner that lends itself to controlled and convenient instrument development and data entry. It is still a further object of the invention to provide a general means of processing financial data that is directed at the macro structure of a financial instrument but may be applied 25 without variance for individual instrument characteristics.

To accomplish these and other objectives of the invention, we have developed a system that employs valuation independent, well-defined financial components (also referred to as financial events) that can be combined to build new financial structures. "Valuation independent" means that financial instruments are modeled independently of their valuation methodologies. A general purpose software model is provided and implemented in this system for representing the structure and characteristics of these products. A declarative specification language is provided to describe financial instruments in a consistent manner that lends itself to processing in such an object oriented system. A general traversal process is provided that can be applied to the macro structure of a financial instrument to implement various functions that produce results based on such information, such as the stream of financial events associated with the instrument, or the pricing or valuation of the instrument. We also provide within this system a framework for processing the resulting structures in a way that supports various valuation methodologies. Techniques including double dispatch and other mechanisms are utilized to provide flexible means of associating the appropriate processing methods with the diverse range of instrument characteristics that are encountered in a typical financial institution's course of business. Such techniques allow developers to quickly and easily incorporate new valuation methodologies into the system as these methodologies become available.

We will describe the basic concepts of these models in a way that is as implementation independent as possible; hence this specification will contain pseudo-code only when necessary to illustrate the concepts in question.

This design has been implemented in Smalltalk and Java and we believe it can be implemented in any other object oriented language. Of course, since each development environment has its own tradeoffs, the implementation of such a complex system in a new development environment will present challenges, but ones we believe to be within the competence of persons reasonably skilled in the art.

The manner in which the foregoing objectives are attained is further shown by the drawings enumerated below, and the accompanying detailed description. It should be apparent therefrom that although the invention has been illustrated with a particular preferred embodiment (and certain variations thereon), its principles could equally well 5 be implemented in other ways without departing from the scope and spirit of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram showing the component design approach implemented in the present invention.

10 Figure 2 is a diagram showing the micro structure of a financial instrument implemented in accordance with the invention.

Figure 3 is a block diagram showing an example of the macro structure of a financial instrument.

15 Figure 4 is a block diagram showing as an example the structure of a simple equity option.

Figure 5 is a block diagram showing the relationship between the parameter representation of a financial instrument and its event stream representation as generated by an event extraction transformation.

20 Figure 6 is a block diagram showing the use of alternative instrument parameter instantiation for purposes of relational or object oriented persistent storage.

Figure 7 is a block diagram showing an overall view of the structure and processing of the present invention, including the static and event representation of financial instruments; the event extraction transformation process that generates the event representation from the static representation, and the stream processing objects that act upon 25 the event representation to provide the desired business results and data.

Figure 8 is a block diagram illustrating the event stream associated with a simple swap instrument.

Figure 9 is a block diagram illustrating the event stream associated with a simple option instrument.

5 Figure 10 is a block diagram showing an example of a class hierarchy that might be used to implement processing objects.

Figure 11 is a diagram showing certain methods implemented on event and processing classes.

10 Figure 12 is a block diagram illustrating a sequence of events involved in a double dispatch mechanism.

Figure 13 is a block diagram illustrating a sequence of events involved in a nested double dispatch mechanism.

Figure 14 is a block diagram illustrating a swap payment event corresponding to the instrument that was the subject of Figure 8.

15 DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The preferred embodiment of the invention is illustrated in Figures 1 - 14 and Listings 1 - 16 below and described in the text that follows. The reader should bear in mind, however, that the details of this embodiment (and of the variations thereon that are presented) are not intended to limit the scope of the invention.

20 It should be noted that all of the systems described in this specification involve the processing of financial data by means including one or more general purpose digital computers and related storage devices, networks, peripherals, systems and development software. Such processing is accomplished by programming such computers in accordance with the methods and techniques set forth herein.

1. The Main Concepts

1.1. Independent Financial Product Modeling

As stated earlier, the essential concept that underlies all of our work in this area is that financial products can and should be modeled independently of their valuation methodologies. We wish to avoid the situation where the software model of a product is dictated by its valuation methodology.

Valuation techniques are in constant flux, evolving and changing at a rapid pace. For proper maintenance of any complex financial system, it is crucial that the product models not be modified when new pricing algorithms arrive. Instead, one should be able to associate new pricing algorithms with pre-existing financial products without affecting the product themselves.

To illustrate this point, consider a simple equity option. A value for this financial product can theoretically be calculated in several different ways. One could use the classic Black & Scholes formula, a probabilistic tree model or even a Monte Carlo simulation. Each methodology calculates the option's value in a different way and hence one would expect slightly different results. However, even though the values differ, the structure and characteristics of the option do not change. It should be possible to use *any* of these valuation models on the exact same representation of the equity option. To accomplish this we require that the product model and valuation methodologies be independent of one another.

1.2. Financial Component Architecture

Another key concept in this model is a component based approach in which financial instruments are composed of basic financial building blocks. These building blocks are referred to as financial components or financial events.

Here the assumption is made that new financial instruments differ mostly in the *way* in which they compose and organize these building blocks and only rarely in the building blocks themselves.

Each building block models a very well defined and specific financial concept.

5 Each financial component should be very simple and will:

- capture the financial concept it models
- be re-useable by all developers requiring that type of financial concept

One can think of this as a “Lego blocks” approach in which a financial developer has a toolbox of financial building blocks, called “financial event templates” (or sometimes referred to as “financial component templates”) that can be utilized to make whatever financial structures are necessary.

1.2.1. Financial Component Micro Structure

As seen in Figure 1, the toolbox consists of financial component templates, 101, 102, etc. Each financial component can have internal instance variables that point to either 15 simple state information (e.g. numbers and dates) or other components. This means that components can be easily nested to whatever level necessary.

It is important to point out the subtle difference in terminology here between financial events and financial event templates. Templates are used in descriptions as place 20 holders for real events. As will be described later, these descriptions are used to generate real events from the templates.

Each component may have internal variables that point to other components. For each variable that can point to another component an interface must be specified. Any component pointed to by that variable must support the defined interface. The interface must also be “processor independent”. This means that it returns results that will not 25 change depending on the kind of processing taking place.

The association of an interface for all internal variables that can contain other components defines the “micro structure” of a component. This is illustrated by the example of a micro structure for a payment component shown in Figure 2. Note that the processing interface shown in this figure is required of all basic components.

5

1.2.2. Financial Instrument Macro Structure

Given a set of basic financial components, the next step is to provide a mechanism for specifying how different components fit together to form the model for financial instruments. This is represented in Figure 1 as the “Instrument Specifications” section 110.

10

The financial instrument specification describes which kind of components an instrument will contain and how they will relate to each other. This is the “macro structure” of a financial instrument. An example of a macro structure is shown in Figure 3.

15

The important part of the macro structure is how components are related or nested within each other. The macro structure also defines which actual nested components are used within the confines of the micro structure for each component.

1.3. Standard Mechanism for Defining an Instrument Specification

20

At this level of detail, the instrument specification is a very generic concept. It simply means that a standardized mechanism has been defined to specify how financial building blocks are to be composed to represent a given financial instrument. This can be also be thought of as the description of the macro structure of the instrument.

25

There are varying ways to implement this concept and this point will be discussed later. Regardless of the implementation, the critical element for this concept to work is that all developers use the same mechanism to define these instrument specifications. This allows any developer to easily read and understand an instrument defined by a fellow developer. This also creates an environment in which financial instrument developers can communicate, interact and share complex ideas with ease.

1.4. Formalized Financial Instrument Structure

In this model the software structure of any financial instrument is factored into distinct portions. This factoring is done based on underlying financial considerations. Essentially, each product is composed of the following pieces:

5

- *instrument parameters* - a set of state information
- *instrument specification* - a description of the structure of the financial events
- *instrument event stream* - the financial events resulting from the parameters and specification

A financial instrument's object structure is illustrated in Figure 4. The instrument
10 instance holds onto an instance of each piece of its structure. It is important to recognize that *every* instrument, regardless of its type or specific details, has this structure. It is the specialization of the internal instances that differentiates one instrument from another.

This factoring allows for a well defined dependency between the different portions of the instrument. We can represent this dependency as follows:

15 $instrument\ parameters \oplus instrument\ specification = instrument\ event\ stream$

The \oplus operation is obviously not a simple addition operation. It is actually a generic transformation process called *event extraction*. Event extraction is generic because the exact same process is used for all financial instruments. To generate the events for a new type of instrument one simply has to define the specification and parameters and
20 pass them to the event extraction process. This process always creates exactly the same events for a given combination of instrument specification and instrument parameters. This process has many more ramifications which will be discussed in detail in later sections.

This factoring also allows developers to inherit state and functionality independently for each piece of an instrument.

1.5. Instrument and Event Processing

As stated earlier, processing is independent from the framework for definition of financial instruments. The term “processing” is used to cover any and all operations that can be applied to a financial instrument and its events.

5 The intention of this model is that all processing functionality interfaces with the financial instruments in a well defined and structured way such that its separation from the instrument definition models remains intact.

2. Financial Instrument Internal Structure

As described above, there is a well defined, standard structure for all financial 10 instruments as well as a generic transformation process called *event extraction*. These pieces of the instrument's structure and their relationships are illustrated in Figure 5.

On the left side of Figure 5 one can see the “static representation” 501 of the instrument. This representation excludes the laid out event streams. On the right side of Figure 5 one can see some examples of financial event streams 502, 503, etc. that could 15 have been created via the event extraction process. This is the canonical “event representation” of the instrument.

Since, by definition, the event representation of a financial instrument can always be generated from its static representation, it is not necessary to make the event representation persistent. This allows the implementation to support various persistence models. 20

2.1. Instrument Parameters

The instrument parameters provide a data storage service, i.e. - a state server, for the particular instrument instance. One can consider instrument parameters as a bag of data which is filled, e.g. like in a spreadsheet. Later this bag of data is associated with a 25 particular instrument. This supports a much more incremental association of data with financial instruments and could be useful in certain work processes.

There is a fixed interface for accessing and setting these parameters. As long as this interface is adhered to, parameter objects can be implemented in arbitrary ways and multiple implementations can coexist in harmony. The benefit of this flexibility is shown in the following example.

5 **Example - Transparent Database Storage Mechanism**

Consider the case in which a system is designed to integrate both a high volume "vanilla" financial business as well as a low volume, structured, "exotic" financial business. In most cases, a relational database technology is chosen for the vanilla business because it is characterized by relatively well defined and stable instrument structures.

10 The exotic financial business, on the other hand, is characterized by the rapid development of highly complex instrument structures, therefore object oriented databases can be better suited. It is typically true that an exotics business uses associated vanilla financial products for hedging purposes. Therefore we would like to be able to store the vanilla products in either database technology.

15 This problem can be easily solved with the instrument parameter paradigm. Essentially, there would be two hierarchies of instrument parameter objects; one that could be made persistent in a relational database and one that could be made persistent in an object oriented database. There would be a subclass in each hierarchy specifically for the given vanilla product. Both classes would support the same interface and would also support the same number and names of variables. Therefore, it would be transparent to the rest of the system, and to the financial instrument itself, what kind of database technology was being used. The actual kind of class needed would be determined by the underlying database requirements. It is also possible to switch, "on the fly", between database technologies as the different instances of parameter objects are essentially equivalent. This is illustrated in Figure 6.

Another important advantage of this mechanism is that the class hierarchy of parameters is independent of the class hierarchy of all the other portions of the instrument. This separates the type of instrument from the state and behavior of the object that represents the parameters.

5 **2.2. Financial Components**

Financial components model simple, very narrow portions of real finance business concepts. These components must be reusable and very straightforward to understand and manipulate. When new financial instrument specifications are defined, developers *must* use pre-existing financial components unless they require a business concept 10 that is not yet implemented. This achieves the composition strategy stated earlier because each business concept will be modeled by one, and only one, software object.

Using one software object to model a business concept allows for a very flexible system environment. Developers can expect the same object, and therefore the same behavior and state information, for a given business object in any financial instrument. 15 Therefore, adding new types of instrument processing functionality can be accomplished quite generically. This concept will be detailed in Section 3 ("Generic Processing Framework") below.

If a development team adheres to this model one can expect the number of financial components for a given financial business environment to level off quickly. Essentially, the total number of business concepts is exhausted quite rapidly and therefore 20 there is no need to develop more financial components. (For example, we found that about 45 basic components are sufficient to model all fixed income derivative instruments at JP Morgan.) Of course, this does not mean that the number of possible financial instruments is limited in any way. Since the instruments are built from organizations of 25 different combinations of the components the permutations are, in effect, limitless. This

organization is formalized via the *instrument specification* which will be detailed in the Section 2.3 (“Instrument Specification”) below.

In general, the most basic business concepts become the base component class for all further specialization. These base classes typically identify and formalize the framework and interface for each business concept. This results in a component object hierarchy that is neatly and clearly divided along business lines. It allows developers to strongly leverage the inheritance tools available in all object oriented environments. The following example illustrates this strategy.

Example - Cash Flow Components

A financial component that is appropriate for just about every kind of financial business is the payment or cashflow event. The cashflow event models the movement of a certain amount of a given currency, in or out on a given date. It must nominally have the following state information:

- **currency** - what type of cash is involved
- **notional** - amount of currency changing hands
- **payment date** - the date on which the transfer occurs

This event should be used in all financial instrument specifications that require a simple cashflow to take place. Clearly, however, there are more specialized payment concepts in the financial world. Therefore, the simple cashflow event is also the superclass for all other kinds of financial payments.

Now consider a payment that is based on an interest rate applied over a period of time. This concept is also quite basic and applicable to most financial businesses. The model for this concept would be a subclass of the simple payment event, it would have to add the following state information:

- **accrual period** - fraction of a year over which the interest rate applies

This interest rate payment is more specialized than the basic payment class but it could still be used in any event micro structure in which the basic payment interface were required as it supports this interface.

2.3. Instrument Specification

5 In general, the instrument specification defines or describes how financial components fit together in a logical structure that characterizes a given financial instrument. One aspect of this is the description of the macro structure concept introduced earlier.

The specification for each financial instrument must detail:

10

- which parameters are required
- which financial components are to be used and what state they would have
- how the components are related to each other; both with respect to time and encapsulation
- how the parameters are related to the components

15 There are numerous ways in which to formalize a mechanism that allows developers to define such specifications. Three of the main strategies are discussed below, along with their associated advantages and disadvantages:

2.3.1. Financial Markup Language (FML)

One can consider the instrument specification concept to be a kind of special purpose language, much like Hardware Definition Language (HDL) (a language used for 20 some time by electrical engineers to define how electrical components fit together to compose complex systems) or Extended Markup Language (XML). FML would essentially be a declarative language with well defined, enforceable syntax and usage patterns. Hence, there would actually be two representations of a specification; the textual representation and the objectified representation. The textual representation would be easily 25 readable by humans and the objectified representation would be much like a language parse tree; optimized for efficient execution. The process of transforming the textual rep-

resentation to the objectified representation would be handled by a parser. We call this the specification extraction process.

5 This is a programming language independent approach for describing financial instruments. It is different from the programming language in which the event extraction and processing frameworks are implemented.

10 This approach provides the most control over the specification development environment. Since the language syntax can be easily enforced via the transformation process, the designer of the system has the ability to decide all aspects of the language. Developers cannot wander from the syntax and hence it will remain standardized and easily decipherable by all who are familiar with it. This scheme can also provide self-documenting aspects to the system that can be leveraged for many other purposes.

15 Another advantage to this approach is that it can provide flexible factoring and reuse of specifications to the developer. It can also clearly describe what parameters are required for a given financial instrument specification.

20 A drawback to this mechanism is that it can take a significant amount of effort to define and develop a new language description. Secondly, the transformation function provided by the parser also injects some processing overhead into the system because the system must, at some point, convert between the textual and objectified representations.

2.3.2. Programming Language Dependent Specifications

25 This approach is similar in concept to FML in that there are still textual and objectified representations of financial specifications and that there is a well defined syntax. It is abbreviated in such a way that a truly independent language is not defined. Instead, the objectified representation of specifications are directly created with the same programming language as the underlying system.

20 The specification syntax is simply a convention built on top of the regular syntax and semantics of the underlying programming language. This reuses the existing com-

piler or interpreter functionality to transform the textual representation of the specification into the objectified language parse tree.

This approach makes the financial specification language dependent on the underlying programming language and therefore limits flexibility when compared with 5 FML. However, it does require less initial investment in terms of development time and resources and still delivers all the benefits of having a financial specification language.

2.3.3. Algorithmic Event Generation

Instead of using a financial specification language one can generate event streams directly using a standard programming language.

10 In this case, there would not be an explicit financial specification. Code would be written that, when executed, would create the events and event streams from a set of parameters. Therefore, the code itself would serve as the event extraction process as well as the implicit textual description of the specification. The structure of the code could be influenced by the software framework so that it would look similar and be readable by all 15 developers.

This strategy has the advantage that it requires even less initial investment while still delivering events and event streams that can be processed in the generic framework described later. This strategy may also reduce the processing overhead associated with explicitly converting between the different specification representations.

20 A major disadvantage is the lack of control of the specification description process. It is critically important that instrument specifications written by different developers be readable and understandable by all. If this is not the case, than the entire premise of the model begins to break down. It is much more difficult, although not impossible, to enforce a coding framework than a well defined language syntax. This is especially true 25 over the long term evolution of a complex system in a high turnover environment.

This approach also limits the factoring and reuse of specifications as well as making it difficult to determine all the required parameters for a given financial instrument.

2.3.4. Summary

5 We believe that all of the above schemes can be made to be successful in the context of defining financial instruments. It is even feasible that multiple specification implementations can coexist in the same system (for example, to migrate a system from one form of instrument description implementation to another).

10 However, the divergence of the specification process over the long term needs to be considered very carefully when deciding on an approach. Therefore, we would not recommend the algorithmic approach over the specification approaches.

A true FML implementation would definitely be the most flexible, powerful and interoperable solution. This is very much in accordance with the current textual, domain specific data description standards emerging in the context of XML on the Internet.

15 (Other domain specific textual description languages examples are: Chemical Markup Language (CML), Bioinformatic Sequence Markup Language (BSML), Open Financial Exchange (OFE), Open Trading Protocol (OTP), Synchronized Multimedia Integration Language (SMIL).)

2.4. Instrument Event Stream

20 The instrument event stream represents the explicit, financial event structure of a given instance of a financial instrument over time. This can be considered the instantiation of the macro structure of the financial instrument. This structure is completely defined by the instrument specification and the instrument parameters. This is illustrated in Figure 7.

25 The event stream representation maintains the full chronological list of all the financial events for the instrument and all of the dependencies between the events. It is

typically this representation that provides the information most critical to financial systems.

An important concept to note is that processing objects can only process the instrument in its Event Stream representation. This encapsulates the processing domain from the instrument specification domain. A processor should therefore be able to process an instrument's event stream regardless of the way in which it was specified.

2.5. Sample Instrument Specifications

Here we will show a sample specification for a simple interest rate swap leg instrument and an interest rate option instrument in a declarative, textual manner. The style and syntax are arbitrary and are only intended to illustrate the specification concept. They are not intended to describe a syntax implementation. A real implementation of a declarative specification language would be more complex than this example.

This specification will be broken down into two main sections: the instrument parameters declaration and the financial components structure.

2.5.1. Instrument Parameters

Interest Rate Swap Leg Parameter Description

The following pseudo-code is an example of the parameters declarations for a simple swap:

```
SwapLegInstrumentParameters = (
 20    var startDate = 1/1/96,
    var endDate = 6/1/97,
    var paymentFrequency = semi-annual,
    var currency = USD,
    var rateIndex = 3MonthLibor,
    var discountOn = LiborCurve,
    var dayCountConvention = 30/360
 25  )
```

Listing 1 - Swap Leg Instrument Parameters

The first four of these parameters are self explanatory. The `rateIndex` describes the type of interest rate that the payment is based upon. The `discountOn` variable describes which interest rate term structure is to be used to discount future cashflows. The `dayCountConvention` describes how to count the days in the interest accrual period.

5 The values assigned to the parameters are default values. It is assumed that these values can be changed to describe the differences between two instances of the same type of instrument.

Interest Rate Option Parameter Description

10 The following pseudo-code is an example of the parameters declarations for a simple interest rate option stream:

```
OptionInstrumentParameters extends SwapInstrumentParameters {  
    var strikeRate = 5%  
}
```

Listing 2 - Option Instrument Parameters

15 This instrument parameter declaration inherits from the swap instrument parameters and simply adds an additional parameter, the strike rate variable.

2.5.2. Financial Components Structure and Relationships

Interest Rate Swap Leg Specification

16 The following pseudo-code is an example of a specification for a single leg of an interest rate swap:

```
SwapLegFinancialSpecification = (  
    IterateDatePeriodFor  
    (startDate, endDate, paymentFrequency) {  
        rate = RateTemplate {  
            date = period.start,  
            index = rateIndex)  
        accrual = YearFractionAccrualTemplate {  
            start = period.start,  
            end = period.end,  
            rate = rate,  
            dayCount = dayCountConvention)  
        payment = PaymentTemplate {
```

```

        payDate = period.end + 2 days,
        currency = currency,
        accrualPeriod = accrualPeriod)
}

```

5

Listing 3 - Swap Leg Financial Specification

This specification has two main sections. Firstly, there is a date period iterator that represents a periodic repetition of its contents; in this case based on the start, end and frequency parameters.

Within the scope of the date interval iterator three financial components are defined; a rate event template, an accrual event template and a payment event template. Looking closely we can see how the variables for each instance are obtained from the instrument parameters, the iteration period object or an object created earlier in the process. We can tell from this declarative representation of the specification that each instantiated payment event would point directly to, and therefore contain, an accrual event in its accrual period. The accrual event would point directly to, and therefore contain, the rate event. We also find that the payment date is always 2 days after the end of the accrual period.

The amount of information that we can obtain from a simple perusal of the specification is an example of the power of the declarative approach to specification definition. It becomes quite easy to determine the structure of an instrument without having to instantiate it or debug code.

Interest Rate Option Specification

The following pseudo-code is an example of a specification for a single interest rate option stream:

```

25 OptionFinancialSpecification = (
IterateDatePeriodFor
(startDate, endDate, paymentFrequency) {
    rate = RateTemplate (
        date = period.start,
        index = rateIndex)
}

```

30

```

optionRate = OptionRateTemplate (
    date = period.start,
    strike = strikeRate,
    rate = rate)
5
accrual = YearFractionAccrualTemplate (
    start = period.start,
    end = period.end,
    rate = optionRate,
    dayCount = dayCountConvention)
10
payment = PaymentTemplate (
    payDate = period.end + 2 days,
    currency = currency,
    accrualPeriod = accrualPeriod)
}

```

15

Listing 4 - Option Financial Specification

This specification has a similar structure as the interest rate swap leg specification but the rate template has been replace by an OptionRateTemplate that then refers to a RateTemplate. Here the developer is able to reuse the same building blocks for another financial instrument specification. For example, the same YearFractionAccrualTemplate is used with a new type of rate template without any modification. This is allowed because the micro-structure of the accrual event specifies an interface for the rate variable that is supported by both the OptionRateTemplate and RateTemplate objects.

2.6. Event Extraction

As described earlier, the event extraction process implements the \oplus transformation in the formula below.

$$\text{instrument parameters} \oplus \text{instrument specification (objectified)} = \text{instrument event stream}$$

The event extraction process is a generic transformation that is intended to be applied to any specification that is properly formed. It will essentially walk (i.e. - read) the objectified specification in the context of the given parameters and generate the desired event structure.

One very important rule in the event extraction process is that that the event structure is immutable with respect to time. The same process must always generate exactly the same event structure given the same parameter set and instrument specification, regardless of the relative system time. It is the responsibility of the processing object to

5 ignore the events that are not applicable based on any specified date. This significantly simplifies the testing and verification of the event extraction process. The algorithmic event generation option that was described earlier has no explicit event extraction process as the code itself builds the events.

Interest Rate Swap Leg Event Stream

10 If the event extraction process were run on the sample interest rate swap leg specification with the given default parameters, the event structure would be as shown in Figure 8:

15 During the event extraction process, the processing of the date period iterator will create one instance of each component within its scope. This process also supplies each component with a period object that specifies the date interval of the current iteration. The processing of the iterator essentially functions as the date generation portion of the system.

20 Note here that in this instrument there is the concept of a top level financial event collection, the interest payments. From a top level interest payment one can reach all events relevant to that payment. This structural characteristic is particular to this kind of instrument and thus defined in the instrument specification.

Interest Rate Option Event Stream

If the event extraction process were run on the sample interest rate option specification with the given default values, the event structure would be as shown in Figure 9.

25 As expected, the AccrualEvents now point to RateOptionEvents which in turn point to RateEvents. This is an outcome of the specification as described above.

2.6.1. Detailed Example of Event Extraction Mechanism

In the Section 2.3 (“Instrument Specification”) we presented three ways of describing an instrument: (1) Financial Markup Language, (2) Programming Language Dependent Specifications, and (3) Algorithmic Event Generation. Of those, a generic event extraction mechanism can only be applied to the first two, whereas the third, by its very nature, defines the event generation in a procedural way. The same generic mechanism can be used for the first two if both are transformed into a common intermediate representation.

The nature of this intermediate representation is an object graph of spec objects.

For each financial component template, and for all other elements of the instrument description that hold the financial component templates together, there exists a spec object. This spec object holds the state of the financial template. The state of a financial template consists of either simple data types, like integers, floats, dates, etc., or references to variables or events which would be generated during the generic event extraction process.

Examples for such references are: the last payment event, all reset events between two dates, etc. Since this is a description of the event structure, and not the actual event structure itself, such a reference can not be stated by direct reference, but must be stated implicitly via the events collected during the event extraction process.

All spec objects also implement a double dispatching protocol for the event extraction mechanism. Beyond that double dispatching method, spec objects do not define any interesting behavior. All the knowledge on how to extract events is contained in the event extraction mechanism.

For instruments described in FML this implies that an FML parser generates such an object graph. Standard computer science parsing technologies can be applied to generate such a parse tree. For instruments described with a programming language dependent specification, this implies that such a specification would generate this object

graph using programming language specific statements, in essence creating spec objects and "plugging" them together in order to form the object graph.

The intermediate spec object graph needs to be constructed only once for each kind of instrument. It does not contain any information particular to a specific instrument. This information is provided externally, as the instrument parameters, as explained above.

The generic event extractor that generates the "canonical" event structure makes use of the instrument parameterization and the spec object graph for the instrument kind. It double dispatches with the spec objects in the graph and, in that process, creates the appropriate events, collects them, and resolves references described in the instrument specification. Note, this is very similar to the later processing of those financial events, where financial events do not know anything about the valuation method, and all that knowledge is contained in the event processing mechanism.

In the following we discuss the objects and concepts in the intermediate representation, and the generic event extraction process in detail.

2.6.2. Intermediate Representation Objects

The objects contained in the intermediate representation fall into four categories: structural, parameterization, references, and events.

Classes and objects in the structural category describe the relationships of objects in the large: containment, grouping and naming, as well as periodicity. Classes in this category are NamedSpec, NestedSpec, SequenceSpec, and DatePeriods.

State classes describe which aspects of a spec can be parameterized, what the parameterized data for a particular instance looks like, how to represent temporary state, and how to represent the internal state of spec objects. Classes in this category are TempSpec, VarSpec, DataSpec, and FieldSpec.

Reference classes describe objects that are used to form the value of state objects.

All non-literal references form the network of internal relationships between spec objects.

Classes contained in this category are LiteralReference, NamedReference and ScriptReference.

5 Finally, event classes describe the financial components that are to be created and extracted during the processing. The classes here are EventSpec and spec classes for all financial component classes one requires in the model (e.g. PaymentSpec, InterestPaymentSpec, ResetSpec, AccrualSpec, Discount, etc.).

10 An example for an intermediate representation is presented in Listing 5. The specification from Listing 3 including a description of the required variables derived from Listing 1 are shown as intermediate objects. Indenting denotes containment.

```
NamedSpec name = "SwapLegFinancialSpecification"
SequenceSpec
15   VarSpec name = startDate type = Date
   VarSpec name = endDate type = Date
   VarSpec name = paymentFrequency type = DateInterval
   VarSpec name = currency type = Currency
   VarSpec name = rateIndex type = SmileIndex
20   VarSpec name = discountOn type = DiscountIndex
   VarSpec name = dayCountConvention type = Basis

   DatePeriodsSpec iterator = period
   SequenceSpec
25     FieldSpec name = start value = Ref(start) type = Date
     FieldSpec name = end value = Ref(end) type = Date
     FieldSpec name = interval value = Ref(paymentFrequency) type = DateInterval
     ResetSpec sequence= resets
       FieldSpec name = date value = Ref(period.start) type = Date
30     FieldSpec name = index value = Ref(rateIndex) type = SmileIndex
     AccrualSpec sequence= accruals
       FieldSpec name = start value = Ref(period.start) type = Date
       FieldSpec name = end value = Ref(period.end) type = Date
       FieldSpec name = rate value = Ref(resets.last) type = Event
35     FieldSpec name = dayCount value = Ref(dayCountConvention) type = Basis
     InterestPaymentSpec sequence = payments
```

```
FieldSpec name = payDate value = Script(period.end + 2 days) type = Date
FieldSpec name = currency value = Ref(currency) type = Currency
FieldSpec name = accrualPeriod value = Ref(accruals.last) type = Event
```

Listing 5 - Swap Leg Intermediate Representation

5 The intermediate representation of the spec as a whole is wrapped into a NamedSpec which states its name. A NamedSpec is a subclass of a NestedSpec and thus contains an inner spec. In the case of this example that inner spec is a SequenceSpec which groups a number of VarSpecs, followed by a DatePeriodsSpec.

10 Each VarSpec defines a name and an associated type and describes a parameter that is required in order to process the spec successfully.

The following DatePeriodsSpec expresses a repetition of periods and first defines the name of the iterator. It is also a NestedSpec, and since it contains more than one spec in the body, the FieldSpecs and subsequent EventSpecs are wrapped in a SequenceSpec.

15 Each FieldSpec describes a particular part of the state of the DatePeriodsSpec. Each is defined by the name of the state, its value at processing time, and its type. In the case of this DatePeriodsSpec, the value can not be described statically, but are determined at processing time. Their value is determined by the processing context. For all three FieldSpecs in this example the value is a NamedReference to previously declared state, in this case to "start", "end", and "paymentFrequency", all of which were declared via Var-
20 Specs earlier.

The ResetSpec is the first EventSpec inside the DatePeriodsSpec. It, as all the other EventSpecs, consists of a sequence identifier and a number of FieldSpecs. The sequence identifier describes into which named sequence Events generated by the particular EventSpec are added during processing time. A canonical event extractor, which extracts the unmodified and full event structure from an intermediate representation and parameters will thus, for this example, collect ResetEvents into a sequence called "resets",
25

AccrualEvents into a sequence called “accruals” and InterestPaymentEvents into a sequence called “payments”.

The references to assign a value into a FieldSpec use three new concepts in those EventSpecs. The first FieldSpec in the ResetSpec refers to a sub-state of the iterator, previously declared for the DatePeriodsSpec, it refers to the start of the period in each iteration.

The “rate” FieldSpec in the AccrualSpec refers to the last collected ResetEvent in that iteration. Similarly, the accrualPeriod in the InterestPaymentSpec refers to the last collected AccrualEvent. Thus, a canonical event extractor will, in each period, extract three Events, a reset event, an accrual event which points to the reset event, and an interest payment event which points to the accrual event.

The last interesting way to assign a value is via a ScriptReference. The “payDate” FieldSpec within the InterestPaymentSpec describes a value which is a ScriptReference containing: “period.end + days”. This, in effect, is just a pretty print version of a more complex object tree containing a NamedReference and a LiteralReference, spelled out it reads: Script(“addDays”, Ref(period.end), LiteralReference(2)). This script is evaluated during processing and will result in the appropriate value (for all processors which actually create a value for this field in this event).

Two concepts are not illustrated by this example, the nesting of NamedSpecs and the overwriting of declared variables. A NamedSpec starts a new scope for declared variables. Only the variables declared within that scope are visible within the scope, variables declared outside the scope are not visible. Variables declared within a nested NamedSpec are accessible from the outside using a dot notation. This enables a redefinition, e.g. either a renaming, or a value assignment to the variables declared below. Such a redefinition can also happen in the same scope, e.g. a prior declaration always replaces a subsequent

declaration. This is interesting when the body of a NamedSpec is included into another spec, e.g. see Listing 6.

```
15  NamedSpec name = "USD3MonthLIBORLeg"  
  SequenceSpec  
  5   ConstSpec name = currency value = USD type = Currency  
  ConstSpec name = rateIndex value = USD-LIBOR type = SmileIndex  
  ConstSpec name = discountOn value = USD-LIBOR type = DiscountIndex  
  Spec ref = SwapLegFinancialSpecification
```

Listing 6 - Restricted Swap Leg Intermediate Representation

10 Here, the currency, rateIndex, and discount rate are fixed. When this intermediate representation is processed, parameterization of those variables not required and when present their values are ignored.

Another spec can also be included and renamed in the process, this gives all variables a prefix, as if nested in a NamedSpec. This is shown in the example in Listing 7.

```
15  NamedSpec name = "TwoLeggedSwap"  
  VarSpec name = notional type = Float  
  TempSpec name = first.notional ref = notional  
  TempSpec name = second.notional script = "-notional"  
  Spec ref = SwapLegFinancialSpecification as = "first"  
  20 Spec ref = SwapLegFinancialSpecification as = "second"
```

Listing 7 - Renaming in a Swap

Explicitly only one variable, for "notional", is declared in this spec. Implicitly the variables declared in the two included specs are also declared here, albeit with a prefix. Two TempSpecs are used to overwrite the "variableness" of the notional for both included specs, using the dot-notation, the one declares first.notional, the other second.notional. The temp declarations as used here, basically hook the notionals of the included specs together with the notional declared in the TwoLeggedSwap.

2.6.3. Processing the Intermediate Representation

The intermediate representation can be processed in various ways. Analogous to 30 the ideas about valuing financial instruments and the benefits of separating the descrip-

tion of a financial instrument from its processing, the processing of the intermediate representation is separated from the intermediate representation. There is no knowledge in the intermediate representation on how the processing is done, except a generic mechanism that enables the processor to know which object of the intermediate representation is to be processed: the double dispatching methods.

Examples for different spec processors are: SpecPrinter (prints formatted intermediate representation), VariableExtractor (determines which variables need to be specified as spec parameters when processing a spec), EventExtractor (generates the canonical event structure from spec and parameters). Other kind of processing can include: generation of confirms, specialized event extraction for particular pricers.

In the following sections we first describe how the EventExtractor is structured and then how it processes the intermediate representation and generates the canonical event structure.

2.6.4. Event Extraction Objects

The central classes used in the processing and event extraction process presented in Listing 8 (indenting denotes sub-classing).

```
SpecWalker an interface that describes the double dispatch operations all spec
walkers must be able to perform, as well as operations for resolving references.
SpecPrinter a class that can format and print the spec tree and its components.
ResolvingWalker an abstract class that can resolve references.
VariableExtractor a class that computes the required spec parameters.
EventExtractor a class that extracts the canonical event structure.
Bindings an interface to set and access name-value associations
Bindings.UnorderedString an implementation that uses strings and unordered associa-
25 tions.
WalkingContext an interface which describes access to the stack of bindings re-
quired for processing, and reference lookup and setting.
ResolvingContext context used for resolving walker
```

Listing 8 -Event processing class structure

These classes form certain patterns together with the intermediate representation and for some SpecWalkers together with the parameters.

A SpecPrinter provides the simplest example. It takes only the intermediate representation as input. It then makes use of a process called double dispatching to traverse the intermediate representation. Double dispatching is a mechanism that enables us to define all the processing in the SpecWalker, yet have all the knowledge about the particular spec-structure contained in the intermediate representation. This means that one SpecWalker can process all kinds of specs build from the basic intermediate representation building blocks (later in this specification this same mechanism is applied to the processing of the financial events as well).

The protocol defined in Spec, the interface all intermediate representation objects must adhere to, requires that each Spec understands the message `traverseIn(SpecWalker w)`, see Listing 9. Each SpecWalker requires that each implementation of a SpecWalker understands traversal messages for all intermediate representation objects, as displayed in Listing 9.

```
15  public interface Spec{
16      public void traverseIn(SpecWalker s);
17  }
18  public interface Reference{
19      public Object getLiteralIn(SpecWalker w) throws ReferenceNotDefined;
20  }
21  public interface SpecWalker{
22      void traverse(NamedSpec s);
23      void traverse(SequenceSpec s);
24      void traverse(DatePeriodsSpec s);
25      void traverse(EventSpec s);
26      void traverse(VarSpec s);
27      void traverse(TempSpec s);
28      void traverse(FieldState s);
29      void traverse(DataSpec s);
30
31      Object getLiteral(NamedReference ref) throws ReferenceNotDefined;
32      Object getLiteral(ScriptReference ref) throws ReferenceNotDefined;
33      Object getLiteral(LiteralReference ref) throws ReferenceNotDefined;
34  }
```

Listing 9 – Spec, Reference and SpecWalker interface

In each particular implementation of a spec the call to `traverseIn(SpecWalker w)` is bounced back to the SpecWalker immediately, with `w.traverse(this)`, or in the Smalltalk implementation with `walker traverseVarSpec: self`. The compiler takes care about the appropriate method dispatch in Java; that information is encoded in the method name in 5 the Smalltalk implementation.

The implementation for `traverse(SomeSpec s)` in a particular SpecWalker then processes that particular spec. E.g. a SpecPrinter would print the name of the spec, and prints the internal state. A SpecWalker knows about the internal structure of a Spec, e.g. for a NamedSpec, the SpecWalker is allowed to access the name, and it can access the 10 nested spec, it is not allowed to assume though which particular spec is contained as the nested spec. Instead it double-dispatches to that nested spec, using `nested-Spec.traverseIn(this)`, or in Smalltalk `nestedSpec traverseIn: self`. The nested spec then bounces back as discussed before, and the correct spec processing method in the SpecWalker will be called.

15 The traversal of specs does not return any value. Thus references have their own double dispatch protocol which enables the SpecWalker to retrieve the value of a reference within the context of the current processing. A reference can be asked to `getLiteralIn(aSpecWalker)` and will bounce back to the SpecWalker with `aSpecWalker.getLiteral(this)`, or in Smalltalk `aSpecWalker getLiteralNamedReference: self`. Returned is an object which represents the value of this reference at this point of processing. How the reference is resolved and how events are collected is discussed in the following section.

2.6.5. Event Extraction Process

Whereas the SpecPrinter only interacts with the intermediate representation and does not require to keep any processing context, the EventExtract and the VariableExtractor both need to resolve reference in order to compute their value. They both inherit 25 the appropriate behavior from a common superclass named ResolvingSpecWalker.

The ResolvingSpecWalker keeps an instance conforming to the WalkingContext interface (see Listing 10). This is an object which maintains a stack of Bindings and can be asked to push and pop a new Bindings object, to store and retrieve the value for a given name, and to collect and retrieve events.

```
5  public interface Bindings{
6      public static Object UnknownValue = new Object();
7      public static Object InitialValue = new Object();
8
8
9      public void put(String name, Object value);
10     public Object get(String name);
11 }
12
13
14 public interface WalkingContext{
15     public void push(String name, Bindings frame);
16     public void push(Bindings frame);
17     public void pop();
18     public Bindings peekBindings();
19     public String peekNames();
20
21     public void put(String name, Object value);
22     public Object get(String name) throws ReferenceNotDefined;
23     public String getScopeName();
24
25     public Events getEvents(String name) throws ReferenceNotDefined;
26     public Hashtable getEvents();
27     public void collectEvent(String name, Event event);
28 }
```

Listing 10 - Bindings and WalkingContext interface

Every time the resolving spec walker processes a structural intermediate representation object it pushed a new Bindings object on the context. If a NamedSpec is processed the name of the spec is pushed as well. This stack of bindings and names forms the basis for the value lookup mechanism implemented in the concrete implementation for the WalkingContext. Currently one such implementation exists, the ResolvingContext.

When an object is stored into the WalkingContext it is just stored into the top Bindings object. When an object is retrieved via its name from the context and can not be

found in the top binding, the subsequent bindings are searched. For each bindings frame which is named, the name is used as a prefix to the current search name. This implements the overwriting mechanism presented in Listing 7. Note, that the initial name will never be a compound name (using the dot-notation), since the access of names from outer scopes is not permitted.

```
10  public Object get(String n) throws ReferenceNotDefined{
11      Bindings frame;
12      Object result;
13      String name = n;
14
15      String frameName;
16
17      for (int i = frames.size()-1; i>=0; i--){
18          frame = (Bindings) frames.elementAt(i);
19          result = frame.get(name);
20          if (result != Bindings.UnknownValue) return result;
21          frameName = (String)names.elementAt(i);
22          if (frameName != UnnamedFrame){
23              name = frameName + "." + name;
24          }
25      }
26      throw new ReferenceNotDefined();
27  }
```

Listing 11 - Getting an object in the ResolvingContext

When the ResolvingSpecWalker processes a VarSpec, the current value for a reference of that name is retrieved and stored into the context (see Listing 12). If the value can not be found an error situation has occurred, since the parameterization is required to contain all VarSpecs which are not overwritten.

```
30  public void traverse(VarSpec v){
31      Object value = null;
32
33      Reference ref = new NamedReference(v.getName());
34      try{
35          value = ref.getLiteralIn(this);
36      }catch(ReferenceNotDefined e){
37          System.out.println("Error: variable " + v.getName()
38                          + " not defined");
39      }
40  }
```

```
    context.put(v.getName(), value);
}
```

Listing 12 - Traversing a VarSpec in the ResolvingSpecWalker

5 A TempSpec and DataSpec are handled in similar ways, for a TempSpec the reference is retrieved and stored in the context, for a DataSpec, the literal object is simply stored into the context.

When retrieving a NamedReference the ResolvingContext first checks if that reference refers to an event, and if so the event is returned (if a dot notation is used, the correct subpart of the event is returned). For a LiteralReference the value is returned directly since it does not need to be evaluated in a context. A ScriptReference is evaluated by executing the script, using the perform mechanism in Smalltalk, and the reflection mechanism in Java.

The VariableExtractor makes use of this framework by overwriting the implementation for the VarSpec. Its task is to find all required parameters for processing a spec in its intermediate representation form. Instead of raising an error when not finding the value for a NamedReference of the same name, the required variable is collected with the proper prefixed name (if required), and an initialValue is assigned and registered in the current context.

```
20  public void traverse(VarSpec v){
    Object value = null;
    Type type = null;
    String n = v.getName();

    25  Reference ref = new NamedReference(n);
    try{
        value = ref.getLiteralIn(this);
    }catch(ReferenceNotDefined e){
        // register in variables
        30  String scope = context.getScopeName();
        value = Bindings.InitialValue;
        type = v.getVarType();
    }
}
```

```

String k;
if (scope.equals("")) {
    k = v.getName();
} else{
    5      k = scope + "." + n;
}
variables.put(k, value, type);
}
context.put(n, value);
10
}

```

Listing 13 – Traversing a VarSpec in the VariableExtractor

In contrast the EventExtractor can assume that all required parameters are defined. Instead, its focus is the creation of the appropriate Events and their collection into the proper event sequence. In contrast to the VariableExtractor it also needs to handle the 15 iterator object properly.

The event generation for all events is handled in the same way. If it is required by a SpecWalker to treat EventSpecs for different kind of Events differently a sub-interface of the SpecWalker is used which defines protocol for those kinds of EventSpecs explicitly. In the generic EventExtractor an Event is created for an EventSpec by first accessing a 20 Map from EventSpec classes to Event classes. The found class for the particular EventSpec is then asked to create an instance. This instance is then filled with the proper state by iterating over the FieldSpecs. For each such spec the current value for the reference contained in the FieldSpec is retrieved and that state is set into the Event by using reflection (using another mapping from names used in the FieldSpecs to instance variables 25 used in the particular event).

After the state of the event is filled it is stored into the context under the sequence name for that EventSpec. When collecting events the ResolvingContext asks the Event for its preferred Event Collection object. Thus payment events can be collected into a payments collection, whereas reset events are collected into a resets collection. If no preference is given a generic collection is used. Which kind of collection is used is significant for 30

the second step of processing the event structure and e.g. pricing it. The double dispatch mechanism there will make use of the different collections events have been collected in,

The iteration objects (e.g. DatesPeriodSpec) need to be rolled out when extracting events. Analogous to event generation, first a new DatePeriods object is created and its 5 state is filled with values described in the collection of associated FieldSpecs. Based on that object an iterator object is initialized. A context for the iteration is pushed and for each iteration, the next value of the iterator is assigned into the name originally specified in the DatesPeriodSpec. Then the nested spec is traversed using the double dispatching protocol explained above. At the end the context is popped.

```
10  public void traverse(DatePeriodsSpec s){  
11      DatePeriods p = new DatePeriods();  
12      this.setState(s, p);  
13      BluePeriodGenerator it = new BluePeriodGenerator(p.start, p.end,  
14                  null, null, p.interval,  
15                  false);  
16      if (!it.hasMoreElements()) return;  
17  
18      context.push(new Bindings.UnorderedString());  
19      for(;it.hasMoreElements();){  
20          context.put(s.getIteratorName(), it.nextElement());  
21          s.getSpec().traverseIn(this);  
22      }  
23      context.pop();  
24  }
```

25 **Listing 14 - Traversing a VarSpec in the VariableExtractor**

2.7. Benefits of the Financial Instrument Structure

There are several benefits of the financial instrument structure described above that are very interesting from the system design perspective. These are described briefly below.

30 2.7.1. Persistence - Relational vs. Object Oriented Databases

New exotic financial instruments evolve quite rapidly and they are typically characterized by very complex event structures. These event structures are typically quite

diverse. One cannot expect them to be regular extensions of a common structure. An important aspect of an exotic financial system is its ability to incorporate these new instruments and their event structures in a timely manner.

Modern object oriented databases allow complex object graphs to be made persistent. One can design the object model and assume that the database will map the object into a form of persistent storage. The object model effectively "is" the data model so no explicit data model is required. This greatly simplifies the entire development process from the point of view of the developer, especially when the object model becomes quite complex.

10 Relational databases, on the other hand, require not only an object model but an explicit data model and an explicit mapping between these two models. Both the models and the mapping strategy must be kept synchronized at all times for proper persistence behavior to be maintained. This requirement increases the amount of effort and overhead incurred by the business application developers when designing and implementing 15 functionality.

The above discussion points out the major theoretical reasons why complex financial products can be better served by an object oriented database. There is typically more overhead when using a relational database as a persistence mechanism for object oriented systems that depend upon highly complex object models.

20 As stated in Section 2 ("Financial Instrument Internal Structure"), every instance of an instrument can be represented either statically or as an event structure. This flexibility can allow a developer to employ a relational database with far more efficiency. It is now possible to make persistent *only* the static representation of an instance of any given instrument. The event structure can, by definition, always be obtained by transforming 25 the static representation, thereby removing the need to make the event representation persistent.

This benefit is obtained, however, only if one makes the assumption that the parameters in the static representation of an instrument are far less complex than the actual events in the event representation. This assumption seemed reasonable initially and has proven itself to be correct in all implementations to date.

5 One additional benefit is that one can now store instances of financial instruments in more or less constant space with respect to maturity. In many other implementations where the event structure is stored, the space necessary for persistence is linearly proportional to the maturity of the instrument.

2.7.2. Implementation Factoring Benefits

10 The factoring of the financial instrument structure allows a developer to *independently* choose the class (and therefore class hierarchy) for the specification and the parameters. This allows the developer to create a new instrument that has a description (i.e., specification) that is much like an existing instrument, yet create a data binding based on a completely different instrument.

15 2.7.3. Specification Composition

The design also allows specifications to be “used” or “included” by other specifications. This composition strategy gives the developer a mechanism for naturally modeling the fact that, in many cases, new instruments are based not only on similar events but on similar event specifications.

20 The most relevant example of this is in the case of an option to enter into an underlying contract. In most cases, the specification of such an instrument is simply the specification of the underlying instrument plus more information. This mechanism allows the developers to re-use the underlying specification as a part of the new specification.

3. Generic Processing Framework

25 As stated numerous times above, one of the key design goals was to separate the description of financial instruments from the processing of the instrument. Processing

objects take the event representation of an instrument, i.e. its macro structure, and operate on the events and parameters to compute some output value or values. This was done to allow development of new instruments to proceed orthogonally to the development of new processing functionality.

5 It is very important to note that in this design the valuation operation, or pricing, is simply a specific type of processing. In terms of trading and risk management, pricing is the most critical operation. So it will typically be the most visible type of processing, but a type of processing nonetheless. This means that the general framework for both pricing and other kinds of processing objects will be exactly the same.

10 *Subj* The design of this part of the system implements a framework that allows processing objects to walk over an instrument's financial event representation and implement an operation for each and every kind of event that it encounters. The concept is that the total processing algorithm is implemented via proper operation on all of the financial events. This is very much analogous to the concept described earlier; that financial instruments can be created by defining the structure and relationships of base financial components. That is, if we can define instruments via composition of basic components then we should also be able to define processing algorithms composed of operations on these basic components.

3.1. Basic Object Oriented Processing Framework

20 The goal of this basic object oriented processing framework is to supply the tools necessary to implement event based processing for any purpose, be it pricing, processing, querying etc. This framework, once completed, is then leveraged to implement frameworks for the different types of functions. An example of a possible class hierarchy is depicted in Figure 10:

25 As seen in Figure 10, the example hierarchy initially divides into sub-hierarchies that provide frameworks for product pricing 1001 and generic event processing 1002.

The pricing framework then divides into frameworks 1011, 1012, etc. to support different types of pricing. It is important to note that the types of pricing defined are based on pricing methodologies instead of actual product types. This differentiation is necessary to provide the flexibility to define many different pricers for a given instrument type.

For example, it is true that one could price an interest rate swap instrument using multiple methodologies; a cashflow present value approach, a probabilistic tree based approach and a Monte Carlo approach. This framework easily allows for multiple implementations of pricer classes for this type of instrument and also provides the necessary framework for each implementation.

The processing framework provides a generic way to walk over an instrument's macro structure and extract useful information. For example, one could implement a processor that would in effect "query" every product macro structure for all option exercise related events. This processor would be applicable to every existing instrument as well as all future instruments as long as the set of events with option exercise characteristics were consistently defined and utilized. This significantly reduces the amount of maintenance effort required to introduce new instruments as well as increasing the control aspect of critical event processing.

The responsibility of the top level *BasicProcessor* class in this example is quite simple. In this class a *valueEventType: anEvent* method is defined for each an every event type that exists (where *EventType* is replaced with the name of the event class). The implementation of this method for each event effectively registers this event type with the processing system. This means that as developers add event types the only initial requirement for processing is that a method be entered on this class for that event type.

Also, a *valueGenericEvent: anEvent* method is defined. The event specific valuation methods described above are implemented to immediately call this generic method.

Hence, initially, this framework provides that any event will, by default, be processed as a generic event. The implementation of the *valueGenericEvent: anEvent* is a subclass responsibility.

If a developer wants to handle an event in a non-generic manner in any processing sub-class he must simply override the initial implementation of the event's registered valuation method.

The *BasicProcessor* class also implements a *valueEvents* method. This method, when invoked on an instance of a processor, should return the "value" of the instrument when invoked. The implementation of this method is deferred to the sub-classes of the *BasicProcessor* class and what the "value" is can vary widely.

Therefore, the top level framework provides the following features to developers:

- A registration service that defines one valuation method name for each event type
- A mechanism by which all event types, unless otherwise overridden in the subclass implementation, are treated as generic events
- A default event valuation method that is meant to be overridden by each subclass so that default processing behavior can be easily implemented by processor developers
- An API, *valueEvents*, that is the main entry point for initiating a valuation on an instance of a processor.

The combination of the way in which the *valueEvents* method and all event specific *valueEventType: anEvent* methods are implemented amounts to the definition of the processing action of any given processing sub-class.

3.2. Micro Structure Processing

As described earlier, for each specific type of financial event, a processor defines a method which computes a result (i.e. *valueEventType: anEvent*). The type of result, for the given financial event, is purely dependent on the type of processor in question.

For example, a processor that is used to obtain the "price" or MTM of a financial instrument would calculate results for financial events that would be combined to pro-

duce this price for the instrument. A processor that was responsible for querying the instrument to determine all payment flow dates for the next N days would return quite different results for each financial event. It might even ignore some financial events that were not relevant to this operation.

5 For each specific type of financial event, a processor can only rely on the micro structure of that event. It cannot assume a specific type for any nested financial event, only that the nested events adhere to the stated interfaces. These interfaces can be used to provide processor independent information about the internally nested events.

10 This can be illustrated by the specification examples detailed earlier of the interest rate swap leg and the interest rate option instruments. The major difference between these two instruments is that the accrual event points to a different type of rate event. In the former case it points to a basic rate event and in the latter case it points to a rate option event. This is allowable because the micro structure of the accrual event expects a "rate interface" in the `rate` instance variable of the accrual event and both types of rate 15 event satisfy this interface.

All processor dependent results are obtained via a double dispatch mechanism which encapsulates the processor and separates it from the actual type of the nested financial events. This mechanism chooses the next processing step inside the processor for a nested financial event based on the actual event type.

20 3.3. Macro Structure Processing

The actual event type that is nested within any financial event is completely determined by the macro structure of the instrument being processed. As described earlier, the macro structure of an instrument defines the actual events and their relationships to each other. It is this macro structure which determines the overall processing of the instrument. Thus, a processor does not require, and is not permitted to have, any a priori 25 knowledge of the structure of a given financial instrument.

This is a very important concept as it allows a processor to process any well formed macro structure because the macro structure itself drives the sequence of processing steps.

This is best illustrated by an example. Suppose a processor inquires of a given 5 macro structure for its payment events. Only the macro structure has the knowledge of what kind of payment event types are actually contained in this set. Upon receipt of these events the processor visits each event and uses the double dispatch mechanism to choose the right method for processing the actual event type.

When a method is chosen to process a given payment event, that event is then 10 processed as described in Section 3.2 ("Micro Structure Processing"). While in this processing mode the double dispatch mechanism is used again whenever a processor dependent result is required for an event nested within the payment event. This again causes the proper method to be selected based on the actual event type in question and this event type is defined by the macro structure.

15 Therefore, the actual methods and the order in which they are visited on the processor object is completely determined by the macro structure. In this way we say that the macro structure drives the processing of a financial instrument.

In summary, a processor just defines what to do for each specific event type and the macro structure determines which methods are visited and in what order.

20 3.4. Basic Description of Double Dispatch

As described earlier, a double dispatch mechanism is employed to determine which processing method is chosen for a given financial event within a given processor. The example below will illustrate how a double dispatch design pattern is used to implement this behavior.

25 Figure 11 depicts the methods that need to be implemented on the event classes and on the processing class to effect a double dispatch paradigm. The method imple-

mented on the event classes, *valueEventInProcessor: aProcessor*, is meant to be polymorphic across all event classes. These methods essentially define a mapping between the class type and the type of processing that it expects to receive within a processor. It is crucial to understand that these methods do not specify anything about what the processing actually accomplishes, they simply specify which category of processing should be applied to the instance by the processor.

The methods implemented on the processor class are meant to explicitly define the processing algorithm for each type of financial event. It is important to realize that these methods make absolutely no assumptions as to where the event sits in a given macro structure, they simply process an event of the given type based on the event's micro structure. Note that this does not mean that external information, such as market data, cannot be used as a context when processing an event. It is the relationships of the events (i.e. the macro structure) that cannot be assumed by the processor, all other contextual information is allowable in this strategy.

In Figure 12, one can see an example of how an instance of an interest payment event is processed in a given processor. Assume that we are in a processor method for a given event type. This method requires a processor dependent valuation of a nested event. At this point, the nested event type is not known beyond the interface declared in the micro structure for the enclosing event.

The processor sends the message *valueEventInProcessor: self* to the nested event. As described earlier, all events must support this interface for proper processing. The event responds by sending the appropriate message back to the processor based on how this method is implemented for its class. In this case, an instance of an *InterestPaymentEvent* will send the message *valueInterestPaymentEvent: self* back to the processor instance. At this point, the processor knows the actual type of the nested event and can proceed with processing according to the micro structure of that event. Thus, the previ-

ously nested event now becomes the event being processed. Once again, the processor has no knowledge of and should make no assumptions on how the event is embedded in the macro structure of the instrument.

This process occurs recursively within any event if there is a need for processor 5 dependent valuations of its nested events.

3.4.1. Nested Double Dispatch

Nested double dispatch is a mechanism that allows the developer to process any event contained in its micro structure via the double dispatch mechanism. This nested process allows this to occur any number of times. Figure 13 represents a nested double 10 dispatch.

The nested double dispatch simply allows the double dispatch mechanism to be used at any point in the valuation of any other event. The value of the nested dispatch is returned to the calling method and used within the context of the valuation of that event.

3.4.2. Detailed Description of the Pricing Framework

15 Because valuation/pricing is such an important aspect of the modeling and implementation of financial instruments this section will give more detail on this topic.

3.4.2.1. The Cashflow Valuation Methodology Implementation

The implementation of the cashflow valuation methodology is a good example 20 because the *BasicCashflowPricer* class implements a framework which, when applied to a given instrument's macro structure, defines how the macro structure is processed.

Essentially, this class implements a framework that causes all cashflow events to be processed at the top level. A present value is calculated for each cashflow event found in an instrument's macro structure. The value of the instrument is defined to be the sum 25 of all these present values. The code in Listing 15 illustrates this framework:

Implemented Behaviour

```

Cashflow Pricer Class
Instance side:

valueEvents
sumCashflows = 0
self.instrument.events do: [ :anEvent |
  sumCashflows = self.presentValue(self.valueEvent (anEvent))
]
return sumCashflows

-----
valuePaymentEvent(thePayment)
return thePayment.notional

-----
valueGenericEvent (theEvent)
return 0.0
-----
```

Listing 15 - Basic Cashflow Pricing

The *valueEvents* method is overridden from the super class to walk over all events that exist in the instrument, obtain the present value for each value and sum the resulting values. The value of each event is determined by its type by the double dispatch mechanism. Therefore, if an event is a cashflow event the *valueCashflowEvent: anEvent* code is executed and returns the notional value of the cashflow. If the event is of any other type, the framework described earlier forces these events to be valued as generic events. In this case, the cashflow pricer has defined the generic event method to return zero. Hence, only the values of cashflow events will be included in the final summation.

In this manner we have created a processing function by simply summing up the valuation methods implemented for given events. This framework can then be easily extended by sub-classes by simply overriding the *valueCashflowEvent: anEvent* to perform other, more complex operations.

3.4.2.2. Extending the Basic Processing Framework

The basic cashflow pricing framework can be easily extended in a sub-class to handle a more complex cashflow event, a swap payment event that was described earlier in Figure 8. A set of events that represents one of these payment is depicted in Figure 14.

5 The additional/overridden methods for the sub-class of the *BasicCashflowPricer* are shown in Listing 16:

```
Fixed Swap Payment Pricer
Instance side:

-----
valuePaymentEvent(thePayment)
return
thePayment.notional*
self.valueEvent(thePayment.accrual)*
thePayment.accrual.rate

-----
valueAccrualEvent(theAccrual)
return (self.endDate-self.startDate)/self.dayCount
```

Listing 16 - Fixed Swap Payment Pricer Methods

One can see here that by changing one method and adding one more method this 10 new sub-class can value a more complex payment event within the framework already implemented. Essentially, this sub-class has defined that the payment event's value will not be the product of the notional, the rate and the accrual period. The value of the accrual period is achieved via nested double dispatch as it valued from within the *valuePaymentEvent* method. This causes the *valueAccrualEvent* method to be invoked and the 15 value returned.

The framework already supports summing all the cashflow relevant events and taking their present value so this is all the new sub-class must implement to effect a significant change in the valuation method.

3.4.2.3. Supporting Multiple Valuation Methodologies

As mentioned several times, there are usually multiple ways in which to value any given instrument. These are divided up into methodologies as reflected by the sub-classes of the *Basic Financial Pricing* class as depicted earlier. Typical methodologies include:

- Cashflow present valuation methodologies – Each valuation relevant event (e.g. - a cashflow) of a product is valued independently using accepted analytics and the present value is taken.
- Probabilistic tree valuation methodologies – A probabilistic tree is built that is used to determine the valuation of the instrument's events.
- Monte Carlo valuation methodologies – A true simulation is built and driven with some inputs to determine the valuation of the instrument's event.

The above is by no means an exhaustive list of available valuation methodologies. The important concept is that one must provide a framework that can support any number of methodologies as they invariably evolve over time.

If it were desired to be able to price a swap instrument in all the above methodologies one would expect to find one class in each hierarchy that would be capable of this functionality. Then, one could choose the valuation methodology at run time by choosing which class of pricer instance to apply to the instrument.

3.4.2.4. Support for Analytics Implemented in Other Technologies

The analytics for any one methodology can also be implemented in different types of technologies. One typically finds that, as the computational resources required for a given analytical implementation increase, the level of implementation language decreases. This is simply because lower level languages tend to provide better performance on the same hardware. Hence, one would expect to see computational expensive implementations of Monte Carlo simulations implemented built in procedural languages such as C or Fortran. This is by no means always the case but is a good rule of thumb.

Therefore, the pricing framework must also provide for interfacing with analytical implementations that may not be in the same technology. This is typically done by providing an interface between the two technologies in which instrument data is passed into the analytics and pricing data is passed back to the main application. The cross-
5 technology interface is simple to achieve. The issue left to resolve is, how does one convert between the object model of the instrument in the main system and the representation required by the analytical implementation?

The answer is that the pricing framework is essentially reduced to a data format conversion function. The framework for each type of external interface would walk over
10 an instrument's events and, instead of calculating values itself, it would simply convert the applicable events to a data format specified by the external interface. Once a framework for each interface is built it becomes much simpler when new analytical implementations are available for that interface type.

3.5. Benefits of Separation of Processing from Instrument Specification

15 *Mark C1* The classic object oriented view is that the definition of an object should encompass its state and its behavior. This design. On the other hand, specifically and explicitly separates what can be considered behavior from the description. The question raised is -
separates what can be considered behavior from the description. The question raised is -
Why is this deemed necessary?

3.5.1. Long Term Design Stability

20 Let us consider the classic case. Consider the instrument specification defined in Section 2.5 ("Simple Instrument Specification"), which describes a simplified version of an interest rate swap leg and an interest rate option instrument. In the classic object oriented paradigm, all processing methods would be implemented on these objects. For example, the swap object would know how to value itself, return a series of known and un-
25 known payments etc.

To implement a properly modular system around instruments developed in this manner, developers typically define an appropriate interface. This interface typically represents all of the necessary functionality that any instrument must provide to the rest of the system to "fit in". The creation of a new instrument class simply requires that the developer implement the well-defined interface properly for the instrument in question. It is unimportant how the developer implements the interface for the instrument as the interface is meant to hide all the internal complexity of an instrument from other objects.

This strategy seems quite reasonable on first consideration. It is a properly "object oriented" approach and simple to implement. In practice, this approach would work quite well during the initial phases of a financial instrument model.

In reality, however, one must consider the viability of such a model well into the later stages of its existence. The simple interface approach begins to fail as it becomes necessary to add more processing functionality to the system as time marches on. It is always necessary that the new functionality be applicable to not only newly developed instruments but to all existing instruments as well.

This means that all existing implementations of financial instruments would have to have the additional interface individually coded by a developer. As the total number of supported instruments grows, this becomes quite a difficult task. Each new interface can require an inordinate amount of effort, especially if the internal complexity of each existing instrument is not well understood by the developer tasked with upgrading it.

Part 2 The separation of processing from the instrument class and a well defined processing mechanism mitigates this problem. If the model is adhered to, one can expect a new processing class to operate on all existing financial instruments as well any new instruments. This means that if the interface needs to be extended one simply writes a new processor. This reduces the extension of the interface to a single development step as opposed to a series of steps that will get more difficult as the system grows.

3.5.2. Pricing Framework Benefits

There is another reason, more specific to valuation models in the financial arena, for the separation of processing objects from instrument description objects.

As stated earlier, one of the major uses of many financial systems is for risk management and pricing purposes. This means that a very important sub-set of processors are the "pricer" classes. These classes process financial instruments, in the context of a set of market data, and return a monetary value or "mark-to-market" (MTM) of the given instrument. This MTM is a function of the instrument itself, the market data and the valuation methodology.

In the financial pricing arena, there are typically several valuation methodologies that can be used to return a MTM for a given type of financial instrument. In many cases, there is not really a "correct" methodology. The methodology to be used is usually chosen based on a trade-off between accuracy and performance. Typically, the more accurate a pricing methodology the more resources are required.

This means that one instrument can be valued in several different methodologies and the decision as to which methodology is to be used can vary over time. The ability to easily choose a pricer object that implements the desired methodology is a very important and powerful advantage of this model. This is not easily done if all of the processing/pricing knowledge is implemented on the instrument itself.

An additional benefit is that pricing objects can share basic behavior via inheritance and that these objects can be used for multiple instruments. Otherwise, if the processing behavior is implemented directly on the instruments there would be a large degree of code duplication.

4. Future Work

The following sections briefly point to areas that we are currently investigating to broaden the use of our model.

4.1. Lifecycle Support of Financial Instruments

The current focus of the representation of instruments in our model is towards valuation and risk management in front-office and mid-office. Clearly, other kinds of processing and thus other kind of attributes are imaginable and desired during the life-time of a financial instrument.

One interesting approach is to think of a financial instrument not as having only one associate specification and parameters, but to see it as a bag of specifications and parameters, changeable over the lifetime of the financial instrument and targeted towards the different demands of those domains.

One example for this approach is an audit trail. Up to the point where a trade is confirmed no audit trail information needs to be kept, the instrument is in the front-office experimental stage. As soon as the trade is confirmed an audit spec and audit parameters are added to the instrument, stating which finance parameters can be changed at all, by who, and also how and where to record the audit trail. The audit trail might actually be recorded right in the instrument itself, as read-only state related to the audit specification, or it might be recorded in a relational database the audit spec refers to.

Another example is the information required for the back office. As soon as an instrument enters the back-office, a back office spec and associated parameters are added and can describe things such as when payments were made and received, what kind of netting agreements affect the current instrument etc.

4.2. Process Specific Event Extraction Transformations

The approach presented in this paper always uses one event extraction process, which produces the “canonical” event structure, as explained above. For certain kind of event processing, it would be beneficial to use a process-specific event extraction process. This would allow processing designers to fine-tune the time and space behavior of event extraction and event processing.

Instead of asking the instrument directly to generate and return an event structure, this request would always be delegated via the processing which is interested in processing the resulting events. It could then decide not to employ the “canonical” event extraction process, but to use a process specific event extraction process instead. The canonical event structure would still be returned in most cases and also serves as an important concept, since a specification should always be written to generate a canonical event structure which resembles the “real-world” structure of the financial instrument most faithfully. Yet, one would also have the opportunity to optimize special processing cases.

For example, the processing of resets only requires the generation of reset events. The processor could use a specialized event extractor which would ignore the other event streams and only instantiate reset events.

Another example is the integration of external processing methods which are not written using the processing methodology presented here. Such processing methods might be using old existing systems, or they might come from other departments, e.g. the research department, which do not use the same modeling and processing methodology. Often the interface to such systems are one or more function calls which require lists of various dates and values which, in the approach presented here, are spread across various events in the canonical event structure of an instrument.

In the current system this is done by generating all events in the canonical event structure and then running a processor which extracts the relevant data from the event structure.

A specialized event extractor for such processing might not even generate events at all, but instead directly collect the appropriate values into arrays which can then be used as arguments to the external processing method.

Both cases show the advantage of have a specification which can be processed to generate appropriate events, over algorithmically generated events. In the latter case one would need to copy and then rewrite the central part of that algorithm to satisfy the different out requirements. Furthermore, this could not be done in an instrument independent manner, as in the specification based approach, but would have to be done for every instrument which would require such processing.

4.3. Separating Default Values from Specifications

In its current form the default values for variables are contained in the specification itself. This is not suitable for an international usage of instruments. A trader creating a new swap leg in Tokyo expects to see a different initial setting of the state than a trader creating a new swap leg in New York.

One solution would be to create a new kind of specifications for default values. The core of a financial specification would stay the same, but the default specifications would be set based on the user location.

The additional benefit of such an approach would be that the knowledge of inter-related default values could be embedded in such a default-specification. For example, the knowledge to adjust basis and tenor according to a changed currency could be embedded in the default specification instead of in the user interface of the instrument.

5. Conclusion

Our experience shows that the model described here offers significant benefits to the developers of financial systems that must use object oriented technologies to represent financial instruments. As currently implemented, it is biased towards systems where pricing and large scale life cycle processing are the main deliverables. But, we believe that it is flexible and generic enough to be used successfully in various other capacities.

One of the main goals of this project was to design a model that could be maintained and extended for many years to come. Much thought and effort went into the de-